OdeFS: A File System Interface to an Object-Oriented Database

N. H. Gehani
H. V. Jagadish
W. D. Roome

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

Abstract

OdeFS is a file-like interface to the Ode object-oriented database. OdeFS allows database objects to be accessed and manipulated with standard commands, just like files in a traditional file system. No recompilation is required, so proprietary applications can access Ode objects. OdeFS is implemented as a network file server, using the NFS protocol. This paper describes OdeFS and its implementation.

1. Introduction

Ode [2] is an object-oriented database based on the C++ object model [20]. The programming interface to Ode is the O++ database programming language, which extends C++ with facilities for creating and manipulating persistent objects, querying the database, specifying constraints and triggers, and running transactions.

Using O++ to access objects in the Ode database requires writing an O++ program. This can be just as inconvenient as writing a C program for every ad hoc access to a UNIX® file. One solution is to develop a set of interactive utility programs for displaying and manipulating Ode objects, just as UNIX systems have a large set of tools for manipulating files. For example, we could have specialized tools such as an Ode editor, Ode print, etc.

However, it takes a great deal of effort to develop, document, and maintain such programs. Furthermore, it is almost impossible to keep all these tools up to date as the database, operating system, user interface technology, and application needs all change over time.

We propose a different approach. Instead of moving knowledge of Ode into utility programs, we move it into the file system. The *Ode File System,* or *OdeFS* (rhymes with Oedipus), provides a file-like interface to the Ode database. To users, OdeFS looks like part of the UNIX file system. OdeFS has regular files and directories, just like any other file system. However, OdeFS also has *object files.* An object file looks like a regular file, and can be manipulated like one, but an object file is really an alias for an object in an Ode database. OdeFS translates operations on an object file into operations on that Ode object. For example, when a program reads an object file, OdeFS calls a *read* function that the user has provided for the object's class. That read function produces a file representation of the object, which OdeFS then returns to the program as the contents of the file.

Thus commands such as `rm`, `cp`, `mv`, `vi`, and `grep` can manipulate Ode objects, by operating on OdeFS object files. A file-oriented Graphical User Interface (GUI) can display and select Ode objects. Because no code modification or recompilation is required, proprietary applications or applications written in other languages, such as `awk` and Ada, can also access Ode objects.

Consider a group of workstations sharing a Network File System (NFS) [17], and sending requests to a common NFS server. OdeFS is also a network file server, and is interjected between the clients and the standard NFS server. When OdeFS gets a request for an ordinary file, it forwards the request to the standard NFS server and returns its response to the client. When OdeFS gets a request for an object file, OdeFS uses the Ode database to evaluate the object file and returns the appropriate response to the client.

For each Ode class, the user must define a set of O++ interface functions. These interface functions are in a separate class; the existing Ode class is not changed, and can maintain the integrity of its objects. OdeFS calls these interface functions when reading or writing object files for that class of objects.

We assume that the reader is familiar with C++ [20] and the UNIX operating system [6].

1.1 Related Work

There has recently been an explosion of interest in merging the functionality of databases and file systems. Much of this work [1, 8, 9, 16, 18] organizes the data in a text file so that sophisticated database querying can be supported. We have done exactly the opposite. We have structured data in an object oriented database, and would like to be able to manipulate this data with the same ease as files, without sacrificing the structure.

A more closely related stream of research deals with the storage of data in a language and system neutral format, and the support of convenient access through a variety of

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

Proceedings of the 20th VLDB Conference
Santiago, Chile, 1994

languages and other interfaces. The Common Object Request Broker Architecture (CORBA) [14, 15, 19] is one example. The difference is that CORBA implies a new communication protocol between applications programs and databases. To use CORBA, existing application programs, databases, and operating systems must be changed. OdeFS uses an existing distributed file system protocol (NFS), and allows existing file-oriented programs to work on objects, even if those programs were never intended to manipulate objects.

The Shore project [7] also provides a file system interface to an object-oriented database (OODB), but there are many differences. Shore defines a new file system interface dealing with structured files. Shore supports a UNIX-like file system interface as a temporary measure, but they hope that Shore's new interface will eventually displace both byte-oriented files and OODBs. OdeFS is a file system interface to an existing OODB (Ode), and OdeFS mimics the existing UNIX file system interface. OdeFS co-exists with other Ode interfaces, namely O++, `cql++`, and OdeView. We do not expect OdeFS to replace unstructured byte-oriented files; we realize that many tools exist for manipulating such files, and OdeFS allows those tools to manipulate objects as well. Finally, Shore requires the file representation of each object to be precomputed and stored in a designated field. OdeFS does *not* require file representations to be materialized; instead, OdeFS calculates them dynamically, on demand.

2. Design Goals And Decisions

2.1 Goals

The fundamental goal of OdeFS is to allow existing file manipulation programs to manipulate objects. To see why this is important, consider Figure 1, which illustrates the conventional approach to manipulating Ode objects. The Ode utility programs would be written in O++. They would be similar to existing programs for files, but they cannot be identical, so we would have to write and maintain them for Ode.

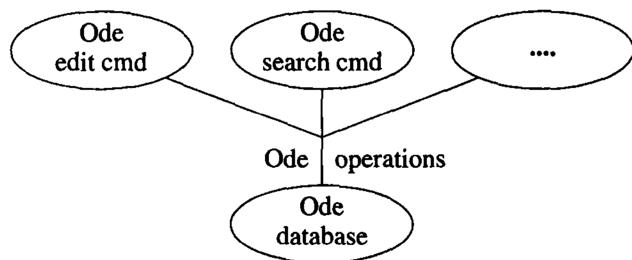

Figure 1: Conventional Approach To Object Manipulation

Figure 2 illustrates the OdeFS approach, using standard file manipulation programs. When run on files in OdeFS, the operating system directs those file operations to the OdeFS server, which translates them into O++ operations on the Ode database. Other O++ programs can access the Ode database in parallel with OdeFS.

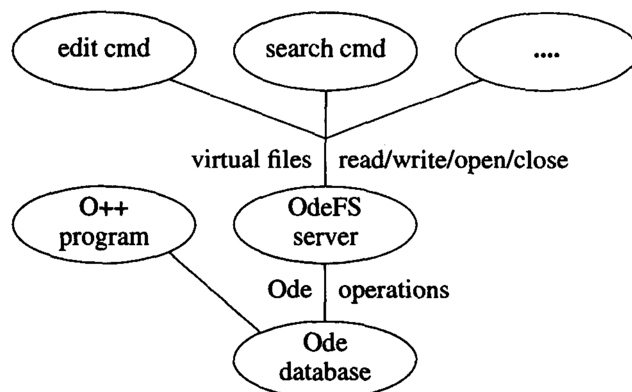

Figure 2: OdeFS Approach

To see the advantage of OdeFS, suppose we have a large, popular, file manipulation program, such as a text editor, and we want to use it to manipulate objects. With OdeFS, we can simply run the program: anything that manipulates files can manipulate objects. The language the program is written in is immaterial. Without OdeFS, we would have to rewrite the program, turning file references into O++ operations, and then maintain the rewritten version.

Another goal is that OdeFS should be object-compatible with the other Ode interfaces. That is, OdeFS should be able to manipulate the objects created with the other Ode interfaces. These include O++ [2, 3, 5, 12, 13], a programming interface based on C++, `cql++` [11], an interactive SQL-like interface for the relational database user, and OdeView [4, 10], a user-friendly graphical interface for the non-programmer.

2.2 NFS

We have implemented OdeFS as a network file server, using the NFS protocol [17]. As in Figure 3, client computers treat OdeFS just like any other NFS server. A client computer's kernel translates system calls on OdeFS files into NFS requests to the OdeFS server.

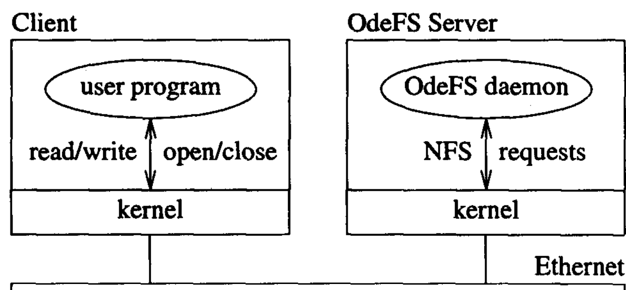

Figure 3: OdeFS as NFS Server

To be precise, when a program on a client computer issues a system call for an OdeFS file, the client's kernel translates the call into an NFS request message, sends it to the OdeFS server, and waits for a reply. The OdeFS daemon process does the request and sends a reply to the client. This daemon process uses standard services provided by the operating system; no kernel modifications

```
class person {
    char   lname[32], fname[20];       // last & first names
    char   addr[60];                   // address
    persistent person* spouse;         // -> spouse
 public:
    person(const char* name);          // parse name as last,first
    const char* getName();             // return last,first name
    const char* getLname();            // return last name
    const char* getAddr();             // return address
    persistent person* getSpouse();    // return spouse (or 0)
      // plus functions to get and update various fields
    };
// Create object for John Smith and set address:
    persistent person* p = pnew person("Smith, John");
    p->setAddr("123 Main St");
// Print all "Smith" objects:
    for (p in person) suchthat (strcmp("Smith", p->getLname()) == 0) {
            cout << "Name    " << p->getName() << endl;
            cout << "Address " << p->getAddr() << endl;
            if (p->getSpouse() != 0)
                cout << "Spouse  " << p->getSpouse()->getName() << endl;
    }
// Remove all "Smith" objects:
    for (p in person) suchthat (strcmp("Smith", p->getLname()) == 0)
            pdelete p;
```

Figure 4: O++ operations on persistent objects

are required. To the client's kernel, the OdeFS server looks like any other NFS server; to the OdeFS server's kernel, the OdeFS daemon looks like any other process.

The advantage of implementing OdeFS as a network file server is that it can run as a user-level program, and multiple clients can easily access it. The advantage of using an existing protocol, such as NFS, is that many computers already use it; we don't have to change the kernels of the client computers. We chose NFS over other protocols because it is simple and is widely used. There are some disadvantages to using NFS; see Section 8.8.

3. Overview Of Ode and O++

O++ is C++ extended with persistence: objects may exist between invocations of programs. Figure 4 shows a `person` class, and uses O++ to create and manipulate persistent `person` objects (the next section will show how these operations are done in OdeFS). First we create a new persistent object, using the O++ `pnew` operator. This operator returns a *persistent pointer,* in this case of type `persistent person*`, and is otherwise identical to the C++ `new` operator. We can use persistent pointers just like regular pointers. In particular, we can use them to call a persistent object's member functions, which Figure 4 does to set the address field.

O++ groups all persistent objects of the same type into a *type extent,* or just *extent.* The O++ `for` statement iterates over all persistent objects in a given extent. The optional `suchthat` clause restricts the loop to objects that satisfy the predicate. Figure 4 uses these constructs to print the name and address, and name of spouse, for all objects whose last name is Smith. Finally, the O++ `pdelete` operator deletes a persistent object, and is otherwise identical to the C++ `delete` operator. Figure 4 uses this operator to delete all Smiths.

4. An Informal Introduction To OdeFS

4.1 Interface Functions

An *object file,* or *ofile,* is an alias for an object residing in the Ode database. OdeFS makes object files look like (hard) links to regular files. OdeFS implements operations on object files by calling user functions defined for the corresponding object class. These functions are *not* defined in the persistent class; instead, they are in a separate *interface* class. The user provides the interface class when adding a persistent class to OdeFS. This interface class is derived from a class provided by OdeFS, which provides default versions of the functions. Table 1 lists the interface functions, and the Appendix gives an example.

As an example, suppose that the user wants to display a `person` object or a program wants to read a `person` object. OdeFS calls the `ofsRead` function of the `person` class. This function creates the object file representation seen by the program, and thus defines the object file format. Our examples use a simple field/value format:

`ofsName`	Return file name for object.
`ofsRead`	Return object file for object.
`ofsWrite`	Update object.
`ofsCreate`	Create new object.
`ofsFind`	Return object matching query.
`ofsLookupRef`	Return referenced object.
`ofsReplaceRef`	Change referenced object.

Table 1: OdeFS interface functions

```
Name    Smith, Sam
Address 123 Main St
```

However, OdeFS does not require this format, nor does OdeFS require object files to be printable. The object file format is completely determined by these user-provided functions, and different objects can have different object file formats. Similarly, when a user or a program updates an object file for a `person` object, OdeFS calls the `ofsWrite` function for the `person` class. This function interprets the data written by the user, and updates the object if the data is valid, or else rejects the write and generates an error message.

4.2 Simple Examples

In OdeFS, an *extent directory,* or *edir,* corresponds to an Ode type extent. The `person` type extent, for example, contains all objects of type `person`. The following commands change to a previously-created extent directory for `person` objects and lists its contents (the `$` is a prompt):

```
$ cd person
$ ls
:class      :database  :newobj
:objects    :src       :tsrc
```

Names beginning with a colon (`:`) are reserved by OdeFS. For example, the file `:class` has the name of the class defined by this extent directory, and `:database` is a symbolic link to the Ode database containing this extent. `:src` and `:tsrc` are directories containing the source code for the interface class for this extent. `:objects` and `:newobj` are *object directories,* or *odirs.* An object directory holds object files (as above). The `:newobj` directory is used to create new objects in the extent. OdeFS creates a new `person` object whenever the user creates a file in `:newobj` and writes to it:

```
$ echo "Name Smith,Sam" >:newobj/sam
```

This creates an object file named `sam` for the new object. We can use ordinary commands to list the object files in `:newobj` and display the newly created object:

```
$ ls :newobj
sam
$ cat :newobj/sam
Name    Smith, Sam
Address
```

The displayed object file is not identical to what we wrote into it. The reason is that when we read an object file, OdeFS calls the associated `ofsRead` function, which in this case inserts white space for alignment, and always displays the address field, even if it is empty. We can specify an address by just writing into the object file:

```
$ echo "Address 123 Main St" >:newobj/sam
$ cat :newobj/sam
Name    Smith, Sam
Address 123 Main St
```

OdeFS passes the user's data to `ofsWrite`. That function parses the data, updates the address field, and does not change the name fields.

The object directory `:objects` has an object file for every object in this extent. The object file names in `:objects` are determined by the `ofsName` function; here the form is *last-name,first-name,* with no blanks. We can list this directory, and use file name expansion to display all the Smiths:

```
$ ls :objects
Gehani,Narain  Roome,Bill
Smith,Ellen    Smith,Sam
$ cat :objects/Smith,*
Name    Smith, Ellen
  ...
```

4.3 Removing Object Files And Objects

In a UNIX file system, a directory entry is really a (hard) link to a file. A file can have any number of links, and all links to a file are equivalent. Removing a link just removes that directory entry. The file itself is not removed until its last link is removed. OdeFS treats object files as links to objects. You can create and remove object files by using the `ln` and `rm` commands. Removing an object file removes a directory entry — a link to the object — but does not remove the object itself. Because the `:objects` directory has an object file for each object, and because users cannot remove (or rename) object files in `:objects`, a user cannot remove all the object files for an object.

OdeFS can delete objects from the Ode database, but with a different mechanism. Associated with every object file is a read-only *pseudo file* representing the object itself. To access this pseudo file, append `:object` to the object file name (we call this a pseudo file because these names do *not* appear when listing the directory). OdeFS deletes the object when the user removes this pseudo file, so the following removes all Smiths:

```
$ for i in :objects/Smith,*
  do rm $i:object; done
```

When OdeFS removes an object, OdeFS also removes all of its object files.

4.4 Errors

A `person` object must have a non-empty name field. This rule is enforced by the `ofsWrite` function for the `person` class. It also rejects any attempt to remove the name. OdeFS then rejects the write request. In this case, the `ofsWrite` function generates an error message, and OdeFS makes it available via the object file's `:err`

pseudo file:

```
$ echo "Name" >:newobj/sam
write error: permission denied
$ cat :newobj/sam:err
Name field required
```

The error pseudo file has the error message generated by the most recent update attempt on that object. The message is cleared by a successful update.

4.5 Object Directories

Users can create their own object directories, by creating an empty directory and either moving an existing object file into it, or creating a link to an existing object file. The following commands create an object directory and populate it with object files for the authors of this paper:

```
$ mkdir authors
$ ln :objects/Gehani,Narain authors/nhg
$ ln :objects/Jagadish,H.V. authors/jag
$ ln :objects/Roome,Bill authors/wdr
```

The link (`ln`) command creates a new object file for an object. Object files are like hard-links to files; updating any object file for an object updates all of its object files. Object files can have any name, and an object can have any number of object files.

4.6 File Name Queries

In an extent or object directory, OdeFS treats a name in the form of a simple query as an object file for the object that matches that query:

```
$ cat ":Address=124 Burlington Rd"
Name    Roome, Bill
Address 124 Burlington Rd
```

The user can treat this like any other object file. Specifically, when the user opens a file name of the form *:name=value*, OdeFS calls the `ofsFind` function defined for that extent, with the file name as the argument. That function interprets the file name as a simple query, locates the matching object, and returns a pointer to it. OdeFS then treats it as an object file. If the query syntax is invalid, or there is no such object, `ofsFind` returns 0, and OdeFS gives the user a "not found" error.

4.7 Inter-Object References

A `person` object contains a reference to a spouse. Assuming `Sam` is an object file, `Sam::spouse` is the object file for Sam's spouse:

```
$ cat Sam::spouse
Name    Smith, Ellen
Address 582 Main St
Spouse  Sam
```

When OdeFS gets an open request for a name of the form *object-file::member*, OdeFS calls the `ofsLookupRef` function provided the class designer. If the name is valid, `ofsLookupRef` returns a pointer to that object, and OdeFS makes it look like an object file. If not, the function returns 0, and OdeFS returns a "not found" error.

The remove and move commands change pointers to referenced objects. (Once more, remember to think of these pointer filenames as links in UNIX). Thus the following commands divorce Joe, and then marry Joe and Nancy:

```
$ rm Joe::spouse
$ ln Nancy Joe::spouse
```

This is handled by the `ofsReplaceRef` function, which is given the member name plus a pointer to the source object (or 0 for a remove request). The function updates the pointer or rejects the request. Incidentally, `ofsReplaceRef` calls the `setSpouse` function of class `person`, which ensures that the spouse relation is symmetric. Thus the remove command automatically sets Nancy's spouse pointer to 0, as well as Joe's.

We regard `Joe::spouse` as another link to the object for Nancy. The remove command removes that link, but does not remove the object (this is how the remove command works in most file systems). Thus removing `Joe::spouse` sets Joe's spouse pointer to 0, but does not remove Nancy's object.

5. Design Overview

5.1 Typed Files and Directories

OdeFS has more complex semantics than a standard file system. To deal with this, OdeFS defines several file and directory types. These types simplify the description of the semantics of OdeFS and control the operations which OdeFS allows on files and directories. These directory and file types are logical types. Client operating systems do *not* know about these logical types: as far as the operating systems are concerned, all OdeFS directories are directories, and all OdeFS files are files.

OdeFS has three types of files. The type of a file is determined by its name and the type of the directory in which it appears. A UNIX file, or *ufile,* is an ordinary file with no particular semantics. OdeFS stores them as ordinary files in an underlying file system. An object file, or *ofile,* is an object alias with file-like representations. OdeFS does not store them as files; instead, OdeFS calls the appropriate interface functions to manipulate the corresponding object when it has to perform object file operations. A control file, or *cfile,* is used to communicate with OdeFS. Users read control files to get information from OdeFS; users update control files (or create or delete them) to give information to OdeFS, or to ask OdeFS to perform various actions.

OdeFS has four types of directories. A UNIX directory, or *udir,* contains UNIX files. An object directory, or *odir,* contains object files. A database directory, or *dbdir,* defines an Ode database to be used by OdeFS. Finally, an extent directory, or *edir,* defines an Ode type extent. A user tells OdeFs about an extent by defining control files in an extent directory. Users get information about an extent by reading control files in the extent directory.

UNIX files are not allowed in object directories, and object files are only allowed in object directories: this restriction helps users distinguish object files from UNIX files.

The type of a directory is determined by a (zero-length) control file in that directory (`..edir`, etc.). A newly created, empty directory is untyped; the user specifies the type by creating one of the above control files. Also, whenever possible, OdeFS deduces the type of an empty directory from the first operation on it. For example, if the user creates an ordinary UNIX file in an empty directory, OdeFS automatically makes it a UNIX directory, and if the user moves an object file into an empty directory, OdeFS makes it an object directory.

OdeFS allows the user to create an arbitrary hierarchy of extent directories, UNIX directories, object directories, and database directories, with arbitrary names. In particular, object directories can be anywhere; they do not have to directly under an extent directory. Also, if class `emp` is derived from `person`, their extent directories can be anywhere; one does not have to be directly under the other.

OdeFS, and users, also need to identify control files. Therefore OdeFS reserves all names that contain a colon (`:`). Also, in any directory other than a UNIX directory, OdeFS reserves all names starting with a period.

5.2 Pseudo Files

Some control files and object files are *pseudo files*. A pseudo file can be accessed explicitly by name, but does not appear in a directory, and does not participate in file name expansion. In particular, pseudo files are not visible when a directory is listed, and cannot be made visible. Pseudo files differ from files whose names start with a period; those names exist, and `ls` will display them if requested.

OdeFS uses pseudo files in two situations. The first is for names that would cause too much clutter when enumerated in a directory listing. For example, if `Sam` is an object file, several pseudo files give additional information about that object: `Sam:object`, `Sam:class`, `Sam:err`, etc. These pseudo files are useful occasionally, but most of the time users do not need them, and do not want to see them.

OdeFS also uses pseudo files for names that are impractical to enumerate. An example is a file-name query, such as `:Name=Joe`, as mentioned in Sec. 4.6. Because such queries are evaluated by a class member function, OdeFS cannot possibly enumerate all valid names of this form.

5.3 Ownership and Protection

In general, OdeFS provides standard UNIX file system permissions for all directories and files. Thus there are separate read, write, and execute permissions, available to owner, group, and other, with each of these being independently settable. Standard commands (`chmod`, `chgrp`, etc.) set these attributes. OdeFS applies additional restrictions to some control files.

OdeFS executes arbitrary functions provided by the user. These functions could have bugs, or could even be malicious. To protect itself, OdeFS creates a separate process for each Ode database that it uses, and calls the interface functions from these processes.

5.4 Objects Known To OdeFS

OdeFS normally provides access to all objects in an extent. Sometimes this is neither possible nor desirable. OdeFS provides two mechanisms that allow the user to control the objects which are known to OdeFS. First, the user can ask OdeFS to *scan* an extent: OdeFS searches the extent and provides access to all objects in it. A user might do this after some other program created objects in an extent. Second, OdeFS can *unlink* an object: OdeFS forgets about that object, but leaves the object in the Ode database. A user might do this to decouple OdeFS from an extent without removing the extent altogether.

6. Detailed Description

6.1 Database Directories

An Ode database can be accessed by multiple programs, including instances of OdeFS — Ode manages concurrent accesses. A database directory defines an Ode database used by OdeFS. It contains the one control file, `:dbspec`, which specifies the Ode database and server.

A database directory also contains an `:extents` subdirectory, which lists every class known to OdeFS in this database. The entries are symbolic links to the corresponding extent directories, and the entry names are the class names. OdeFS maintains this directory, and automatically creates and removes entries as the user adds or removes extent directories. The `:extents` directory is read-only to users; users cannot directly create, remove, or rename files in it.

6.2 Extent Directories

An extent directory describes a type extent. It contains the following control files and built-in directories:

`:class`	A file containing the name of the class of the objects in this extent.
`:database`	A symbolic link to a directory defining this extent's Ode database.
`:newobj`	A built-in object directory for creating new objects.
`:objects`	A built-in object directory with an object file for every object in this extent.
`:src`	The currently used source directory for the interface functions.
`:tsrc`	The trial (''next version'') source directory for the interface functions.
`.compile.b`	When the user creates or touches this file, OdeFS compiles the class.
`.compile.e`	Error messages from the last compilation attempt.
`.flush`	Touching this file causes OdeFS to flush cached information.

`.scan` Touching this file causes OdeFS to scan the database and "link in" all persistent objects of the specified type.

In addition, an extent directory can have any type of subdirectory, and can have arbitrary UNIX files (documentation, etc.).

`:objects` is a read-only object directory containing an object file for every object in this extent. `:newobj` is a writable object directory, with the property that when a new file is created in it, OdeFS creates a new object.

When the user touches `.scan`, OdeFS scans the Ode database for all objects of this persistent class, and links them into OdeFS. Touching `.flush` causes OdeFS to flush any cached information about object file size or contents. These are useful when other Ode programs add or update objects in parallel with OdeFS.

6.2.1 Creating And Compiling An Extent Directory

To create an extent directory, the user creates the file `..edir` in an empty directory. OdeFS responds by creating the `:src` and `:tsrc` directories. The extent directory can have any name, although it is most natural to use the class name.

Once an extent directory has been created, the user must specify the name of the class, the Ode database it exists in, and the source for the interface functions. The user does this by creating the `:class` and `:database` control files in the extent directory, and `:methods.c` in the trial source directory `:tsrc`. At this point, OdeFS treats these as ordinary files. Thus they can be written multiple times, removed and recreated, etc. When ready, the user asks OdeFS to compile the extent directory, by creating (or touching) the control file `.compile.b`. When done, OdeFS creates the control file `.compile.e`, and saves any error messages in it. If successful, OdeFS copies the source files from `:tsrc` to `:src`, and creates the built-in object directories `:newobj` and `:objects`. At this point, the extent directory has been integrated into OdeFS.

To change the interface functions and recompile an extent directory, the user first updates the files in `:tsrc`, and then compiles the extent directory, as before. If successful, OdeFS copies the new interface functions to `:src`. If not, OdeFS continues to use the old interface functions. The user can change files in `:tsrc` at any time, but OdeFS does not use them until the extent directory is recompiled.

Once an extent directory has been successfully compiled, OdeFS does not allow the user to change the `:class` or `:database` control files.

6.2.2 Interface Functions

To access a persistent objects with OdeFS, the user must define an interface class. This interface is derived from the OdeFS class `OfsExtFcn`, and is typically named `OfsExtFcn`_*pclass*, where *pclass* is the persistent class. The base class provides default versions of the interface functions. For example, to implement a read-only OdeFS browser for an existing Ode database, one could just provide `ofsName` and `ofsRead`. The other functions can be added later, as desired.

The update functions can reject a user's update request, in which case they can provide an error message by calling the built-in OdeFS function `ofsErrMsg`. OdeFS makes this message available to the user, via the `:err` mechanism (Section 4.4).

The interface functions include:

`ofsCreate`. OdeFS calls this function to create a new object: that is, when the user creates a new file in the `:newobj` directory. The arguments specify the data written by the user. The function uses `pnew` to create the object, initializes it according to the values written by the user, and returns a persistent pointer for the object. The function returns 0 if it cannot create a new object.

`ofsRead`. OdeFS calls this function when the user reads an object file for an object. The function creates the object file representation for this object in the buffer specified by the arguments, and returns the size of the object file, in bytes.

`ofsWrite`. OdeFS calls this function when the user writes an object file; the arguments specify the data written by the user. The function updates the object appropriately, and returns 1 if successful, 0 if not.

`ofsName`. This function puts the object file name for this object into the buffer specified by the arguments. OdeFS uses this name in the `:objects` directory. The name should be unique within an extent, although OdeFS adds a disambiguating suffix if necessary. If the name is empty, OdeFS will not create an object file in `:objects`.

`ofsFind`. OdeFS calls this function when the user specifies a file-name query: that is, asks for an object file whose name is of the form *:name=value*. The argument is the file name, without the leading colon. If this name specifies an object, the function returns a persistent pointer for it. If not, the function returns 0, and OdeFS returns a "not found" error to the user.

`ofsLookupRef`. OdeFS calls this function when the user accesses a *::member* pseudo file for an object. If the member name is valid for this object, the function returns a persistent pointer to the referenced object. If not, the function returns 0, and OdeFS returns a "not found" error.

`ofsReplaceRef`. OdeFS calls this function when the user moves an object file to the *::member* pseudo file for an object, or when the user removes that pseudo file. If the member name is valid, and the object is of the correct type, this function updates the reference and returns 1. If not, the function returns 0. Note that this function must verify the object type.

OdeFS can handle persistent classes that are derived from other classes. Suggested programming style is that the interface class for the derived class be based on the

interface class of the base class, and use its functions. For example, if class emp is derived from person, ofsRead for emp can first call ofsRead for person, to get the object file representation for the person part, and then appends the representation of the emp fields.

The interface functions described above restrict the maximum object file size to 8192 bytes (the NFS block size). OdeFS can handle larger object files, but the class designer must use a different, and slightly more complicated, set of interface functions. Space does not permit describing them here.

There are some restrictions on these functions. First, ofsWrite and ofsReplaceRef must be idempotent; that is, the resulting value of the object must be the same no matter how many times the function is called (because of timeouts and repeats, OdeFS might get the same request several times, and hence could call these functions several times). This means ofsWrite cannot provide "increment" updates. Second, OdeFS does guarantee that ofsCreate is called only once. However, OdeFS does so by transforming duplicate requests into calls to ofsWrite, so ofsWrite should accept the same data as ofsCreate. And finally, the functions must be reasonably fast: ideally on the order of 10 to 100 milliseconds. If the interface functions take several seconds, OdeFS will no longer feel like a file system.

6.3 Object Directories

An object directory contains an arbitrary collection of objects (object files), all of the same class or subtypes of that class. An object directory has one control file: .extent, which is a symbolic link to an extent directory. All objects in this object directory must be of that type, or a type derived from it. The user can remove or change an .extent in object directory, provided that the object directory has no object files.

Each extent directory has two built-in object directories: :objects and :newobj. :objects has an object file for every object in the extent. This directory is read-only to users, although users can update object files in it.

:newobj is a writable object directory, with the property that OdeFS creates a new object when you create a new file in it. Once created, the object file becomes an alias for the object, and subsequent writes update that object. Creating a file is different from creating an alias; the copy command creates a new file, while a link command creates an alias to an existing object. Thus copying a file into :newobj creates a new object, while linking (or moving) an object file into :newobj does not. Moreover, copying a file into an existing file in :newobj updates an existing object rather than creating a new object.

7. Implementation

OdeFS is built on top of a conventional file system; it acts as a filter between the user and the file system. OdeFS is implemented as several processes (Figure 5).

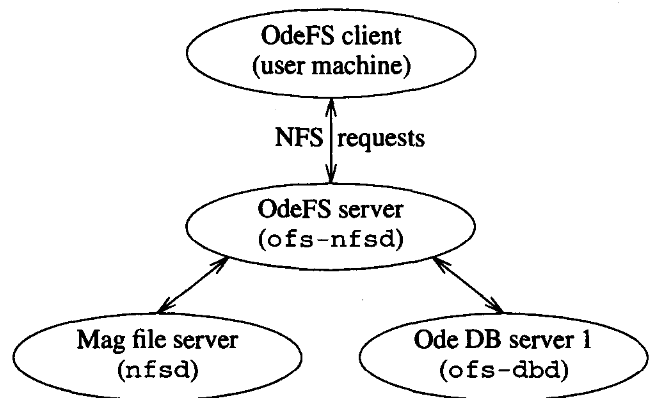

Figure 5: OdeFS process structure

The ofs-nfsd process ("OdeFS NFS daemon") is a user-level NFS server process; it accepts NFS requests from various clients and returns the appropriate replies. ofs-nfsd is also a client of nfsd, the regular magnetic disk file server, and ofs-dbd, a separate O++ program that handles Ode requests. There is one ofs-dbd process for each Ode database used by OdeFS; ofs-nfsd starts these processes as needed. Normally there are several ofs-nfsd and nfsd processes; they act as worker pools, taking requests from a common queue. The ofs-nfsd, ofs-dbd, and nfsd processes are on the same computer. We use the standard NFS interface between ofs-nfsd and its clients and between nfsd and ofs-nfsd. The interface between ofs-nfsd and ofs-dbd is also a request-reply mechanism, but with a different set of requests.

Using a separate magnetic disk file server process simplifies installation and administration, and reuses existing file system tools. Thus most administrative tasks — disk partitioning, formatting, backup and recovery, checking, etc. — are handled by the existing file system.

We have separate Ode database server processes for protection. OdeFS must execute arbitrary, user-provided code; this could be buggy, or even be malicious. All such code is isolated to the ofs-dbd processes. Each runs with the user id of its database, rather than as superuser. The worst that can happen is that buggy user code could damage other classes in its own database, and/or crash its ofs-dbd process. User code cannot damage other databases or the ofs-nfsd process. With suitable timeouts, the ofs-nfsd process can detect such problems and gracefully reject requests to that database and/or restart the process.

All directories, UNIX files, and control files exist as directories and files in the underlying magnetic disk file system. Each of these objects is uniquely identified by the inode number assigned by the file server. For operations on ordinary files and directories, ofs-nfsd just forwards the requests to nfsd, which actually does the work. For operations on control files, ofs-nfsd does the appropriate actions before or after forwarding the

requests to the underlying file system.

Object files are implemented differently: they do not exist as files in the underlying file system. Instead, the `ofs-dbd` process evaluates object files as needed. Each object file is uniquely identified by the persistent object pointer assigned by Ode when the underlying object was created. `ofs-nfsd` forwards object file operations to the appropriate `ofs-dbd` process.

An object directory has two parts: a directory in the magnetic disk file system, and an Ode object, maintained by an `ofs-dbd` process. A directory is really a list of pairs of names and identifiers. This Ode object has the names of the object files and the corresponding Ode object pointers; the directory in the magnetic disk file system has the names of the ordinary files in the object directory. `nfsd` manages the names in the file system part of the directory, and `ofs-dbd` maintains the object file names. `ofs-nfsd` logically combines those two parts and makes them look like one directory. To tie those two parts together, `ofs-nfsd` writes the Ode object directory object's persistent pointer into a private file (`.oid`) in the magnetic disk directory.

As an example, suppose we open, read, and write a UNIX file. The client translates the open request into an NFS lookup request, which it sends to `ofs-nfsd`. The arguments are a handle for a directory, and the name of a file within that directory. The reply is a handle for the requested file or directory, plus its attributes (size, type, update time, etc.). A handle is a capability for a file system object, and is opaque to the client. OdeFS handles contain either an inode number (for files and directories) or an Ode object pointer (for object files), plus some type information. For a UNIX file, `ofs-nfsd` forwards the lookup request to `nfsd`, and returns a UNIX file handle to the OdeFS client. When the client reads or writes that file, `ofs-nfsd` forwards those requests to `nfsd`.

Now suppose that the user opens a control file. In the lookup request, `ofs-nfsd` recognizes that this is a control file, and returns a handle indicating its type. When the client sends a read or write request for that control file handle, `ofs-nfsd` does the appropriate action.

Finally, suppose the user opens an object file in an object directory. `ofs-nfsd` reads the `.oid` file to get the pointer to Ode object directory object, and then sends a lookup request to the appropriate `ofs-dbd` process, asking it to lookup the name in that object directory. That process returns information about the object file, which `ofs-nfsd` then returns to its client. When `ofs-nfsd` gets a read or write request for an object file handle, it forwards the request to `ofs-dbd`.

If OdeFS provides access to a class `person`, then OdeFS must keep some information for each such object. This includes the object's error message and its modification time. The normal technique would be for OdeFS to provide a base class and require class `person` to be derived from it. However, we could not do that for OdeFS, because we wanted OdeFS to work on existing classes *without* changing them. Therefore, for each persistent class known to OdeFS, OdeFS creates another persistent class (e.g., `OfsObj_person`). This is derived from an OdeFS base class, and contains the per-object data that OdeFS needs, including a pointer to the `person` object. OdeFS automatically creates a `OfsObj_person` object for each `person` object. We call these "OdeFS objects" and "user objects," respectively. Internally, OdeFS uses pointers to OdeFS objects. For example, object directories really contain pointers to OdeFS objects.

This two-object approach has proved very helpful. For example, it allows OdeFS to handle classes that do not have a null constructor. The problem is that when a user creates a new object file, OdeFS really gets two NFS requests: a create request, which creates a zero-length file, and then a write request, which has the data for the object. OdeFS must create something for the NFS create request, but unless the user's persistent class has a null constructor, OdeFS cannot create a user object at that time. Therefore OdeFS creates an "unbound" OdeFS object: one that does not yet point to a user object. Then when OdeFS gets a write request for an unbound OdeFS object, OdeFS calls `ofsCreate`, and binds the OdeFS object to the user object created by that function. If `ofsCreate` fails, OdeFS saves the error message in the OdeFS object, leaves the OdeFS object unbound, and calls `ofsCreate` when it gets the next write request.

OdeFS takes advantage of the Ode transaction mechanism. For example, the NFS rename request atomically moves a file from one directory to another. When moving an object file between two object directories, `ofs-dbd` updates those two object directory objects as part of one transaction. Ode then guarantees atomicity.

8. Discussion

Everything described thus far has been implemented. In this section, we describe some open issues. We also explain some of our design decisions and discuss some alternatives that we did not adopt.

8.1 Object Files vs. UNIX Files

In general the user can treat object files and object directories like ordinary files and directories, but there are some differences, particularly with respect to the move vs. copy commands (`mv` vs. `cp`). When you move an object file, the result is another object file for the same object. However, when you copy an object file, the result is an ordinary UNIX file, not an object file. The reason is that the move command uses a single file system operation, which OdeFS intercepts and handles appropriately. However, when a file is copied, OdeFS just sees a series of writes to the destination file. OdeFS cannot tell whether this data came from an object file or a UNIX file or from the keyboard.

To minimize this confusion, OdeFS does not allow users to create ordinary files in object directories. Thus instead

of creating a UNIX file, a simple copy of an object file into a new file in the same object directory will fail.

8.2 Object File Attributes

OdeFS must provide various attributes for each object file: permission, owner, modification time, etc. However, the underlying Ode system does not maintain these attributes for each object, so OdeFS must get these attributes somehow. For simplicity, OdeFS gives all object files in an extent the same owner and access permissions as the extent directory.

However, OdeFS cannot use the same modification time for all object files; OdeFS must provide a reasonable time for each object file. The problem is that most NFS clients cache data, and assume the cached data is valid until the file modification time changes. Therefore OdeFS must change the modification time when the object file representation changes, so clients will invalidate their caches. On the other hand, OdeFS should not change the modification time unnecessarily, because that would disable client caching, which would hurt performance.

Therefore OdeFS keeps an update time for each Ode object known to it, and sets that time whenever OdeFS knows that the object changes. However, OdeFS does not see all object updates. An update to one object can change another as a side-effect. For example, setting the spouse pointer in one object might automatically update the spouse pointer in other objects. Furthermore, an object file can contain data from another object, such as spouse's name. Finally, other Ode programs can update objects independently of OdeFS.

Therefore in addition to the per-object update time, OdeFS also keeps a per-extent and per-database update time. OdeFS sets these times when the user touches the `.flush` control file in an extent directory or a database directory. OdeFS uses the maximum of these as the modification time for an object file.

A better solution would be for Ode to provide a per-object update time, and for OdeFS to add another interface function, which would return a list of objects upon which this object's file representation depends.

8.3 Reading vs. Updating Objects

Using OdeFS for read access to objects is easy and natural. It is easy to browse the Ode object database, and it is easy to write the interface functions. However, using OdeFS to update objects is not quite as easy. The update interface functions are harder to write, because they must cope with arbitrary data written by users, and should provide reasonable error messages.

Furthermore, OdeFS gives delayed feedback for errors. For example, suppose you use a general editor with OdeFS. If you put an invalid entry in a telephone number field, you will not discover the error until you attempt to write the file. A dedicated object editor, on the other hand, would give an error when you changed that field.

8.4 Creating and Removing Objects

Objects are removed in OdeFS by deleting the *ofile:object* pseudo file. One alternative would be for OdeFS to remove an object automatically when the user removes its last object file alias (e.g., treat objects exactly like files). We rejected that idea, because OdeFS does not, and cannot, know about all inter-object references. For example, if employee objects point to department objects, OdeFS should not remove a department object just because the user removes all of the department's ofiles.

Another alternative would be to remove an object (and all aliases) when the user removes its object file from the `:objects` directory. However, we think this alternative makes it too easy to delete objects accidentally.

8.5 One Object, One Object File

An object file represents only one object. In some cases, it would convenient if an object file could represent a number of objects. For example, that would allow a *:name=value* query to return several objects. It would also allow OdeFS to provide a single "all objects" file, with the concatenation of the representations of all objects in the extent.

However, allowing an object file to represent multiple objects complicates both the semantics and implementation of OdeFS, and (at least for now) we have disallowed it. To see the problems, suppose that the file `:Name=Smith` could represent several objects. This implies some convention for separating the objects (a blank line, a row of dashes, fixed length, etc.). We would not want OdeFS to enforce the convention, so we would have to add an interface function to specify a separator string to place between representations. Furthermore, for updates, we would have to add another function to parse the data written by a user into individual object representations. On update, OdeFS would would then call `ofsWrite` for each object. However, this requires OdeFS to match the per-object representations with the original objects, and that means the user could not delete or reorder object representations within a multi-object file.

Of course, we could solve these problems by requiring the per-object representations to be fixed length, or by instantiating a multi-object file as a real UNIX file when it is first opened. However, we do not consider either of those methods to be acceptable.

8.6 Large Extents

The `:objects` directory has an object file for every object in an extent. That will be very inefficient if an extent has 1,000 objects, and will probably fail completely if an extent has 100,000 objects. One solution would be just to eliminate the `:objects` directory for large extents. Users would use the OdeFS query facilities to get object files for the objects of interest.

8.7 Query Facilities

OdeFS provides a simple, single-object query facility (Section 4.6). We are currently building a more powerful query facility. This will allow a `cql++` [11] program, or indeed any arbitrary O++ program, to select objects, and then populate an object directory with object files for them. This relies on the fact that Ode assigns an object id (oid) to every object. An oid consists of three integers, such as (123,5,0), and is the internal representation used for an O++ persistent pointer. OdeFS will treat a file name of the form `.oid=123.5.0` as an ofile for the object with that oid. Then once a program has selected an object, the program can create an ofile for that object just by creating a link from `.oid=1.2.3` to `Sam`.

8.8 NFS

Our choice of the NFS protocol has advantages and disadvantages. The advantages include the fact that NFS is a simple, widely-used easy-to-implement protocol. Also, NFS separates "lookup" from "readdir," which allows OdeFS to have pseudo files.

NFS has a few disadvantages. An NFS client repeats a request if it does not get a response quickly enough. This means that the interface functions cannot have long delays, and must allow duplicate requests (see Sections 6.2.1 and 6.2.2). It would help if an NFS server could tell a client that a particular request will take a while, and the client should use a longer timeout.

NFS clients use the file modification time for cache control, and clients normally cache data they write. However, an OdeFS client should *not* cache data it writes to object files, because OdeFS regenerates them (see Section 4.2). To prevent that, OdeFS plays games with the file modification times that it returns for such write requests. It would be better if the NFS protocol had an explicit mechanism for a server to tell a client to discard any data that is cached for a file.

An NFS server does not know when a program closes a file, so OdeFS cannot tell when a user is done with it. Furthermore, an NFS client can read the contents of a file in any order. These limitations prevent OdeFS from reliably implementing an "all objects" file. Of course, these limitations are why NFS is simple and easy to implement.

9. Conclusion

We have described OdeFS, a file system interface to an object-oriented database system. The major benefit of this interface is that it interoperates well with the UNIX operating system, and thus provides a convenient means for users to access and manipulate objects. The central idea in the implementation is to add a layer of code, between the client operating system and the file system, that implements OdeFS functionality. No modification is need to the front end operating system or the back end file system and object manager. Objects created through OdeFS can be accessed and manipulated by other interfaces to Ode and vice versa.

We are considering several areas of further exploration:

- An "OdeFS interface function generator" tool would be very useful. This tool would read a class definition and create an initial version of the interface class.
- OdeFS should support user-level transactions. That is, a user should be able to update several object files as part of one transaction. Currently, OdeFS treats each update as a separate Ode transaction.
- We would like to allow multiple object file representations for a given object, and allow the user to choose between them.
- OdeFS should allow users to call arbitrary member functions for the object class.

References

[1] S. Abiteboul, S. Cluet and T. Milo, "Querying and Updating the File", *Proc. 19th Int'l Conf. Very Large Data Bases*, Dublin, Ireland, Aug. 1993, 73-84.

[2] R. Agrawal and N. H. Gehani, "Ode (Object Database and Environment): The Language and the Data Model", *Proc. ACM-SIGMOD 1989 Int'l Conf. Management of Data*, Portland, Oregon, May-June 1989, 36-45.

[3] R. Agrawal and N. H. Gehani, "Rationale for the Design of Persistence and Query Processing Facilities in the Database Programming Language O++", *2nd Int'l Workshop on Database Programming Languages*, Portland, OR, June 1989.

[4] R. Agrawal, N. H. Gehani and J. Srinivasan, "OdeView: The Graphical Interface to Ode", *Proc. ACM-SIGMOD 1990 Int'l Conf. on Management of Data*, 1990, 34-43.

[5] R. Agrawal, S. J. Buroff, N. H. Gehani and D. Shasha, "Object Versioning in Ode", *Proc. IEEE 7th Int'l Conf. Data Engineering*, Tokyo, Japan, Feb. 1991.

[6] S. R. Bourne, *The UNIX System*, Addison-Wesley, 1982.

[7] M. J. Carey et al, "Shoring Up Persistent Applications", *Proc. ACM-SIGMOD 1994 Int'l Conf. on Management of Data*, Minneapolis, MN, May 1994.

[8] V. Christophides, S. Abiteboul, S. Cluet and M. Scholl, "From Structured Documents to Novel Query Facilities", *Proc. ACM-SIGMOD 1994 Int'l Conf. on Management of Data*, Minneapolis, MN, May 1994.

[9] M. P. Consens and T. Milo, "Optimizing Queries on Files", *Proc. ACM-SIGMOD 1994 Int'l Conf. on Management of Data*, Minneapolis, MN, May 1994.

[10] S. Dar, N. H. Gehani, H. V. Jagadish and J. Srinivasan, "Queries in an Object-Oriented Graphical Interface", AT&T Bell Labs Technical Memorandum, 1991.

[11] S. Dar, N. H. Gehani and H. V. Jagadish, "CQL++: An SQL for a C++ Based Object-Oriented DBMS", *Proc. of Int'l Conf. on Extending Database Technology*, Vienna, Austria, Mar. 1992.

[12] S. Dar, R. Agrawal and N. H. Gehani, "The O++ Database Programming Language: Implementation and Experience", *Proc. IEEE 9th Int'l Conf. Data Engineering*, Vienna, Austria, 1993.

[13] N. H. Gehani and H. V. Jagadish, "Ode as an Active Database: Constraints and Triggers", *Proc. 17th Int'l Conf. Very Large Data Bases*, Barcelona, Spain, 1991, 327-336.

[14] O. M. Group, in *The Common Object Request Broker Architecture: Architecture and Specification*, OMG, Framingham, MA, 1991.

[15] O. M. Group, in *Object Management Architecture Guide, Second Edition*, OMG, Framingham, MA, 1992.

[16] U. Manber and S. Wu, "Glimpse: A Tool to Search through Entire File Systems", *Proc. USENIX Winter Conf.*, Jan. 1994.

[17] NFS, *Network File System: Version 2 Protocol Specification*, Sun Microsystems, Inc., Mountain View, California, 1988.

[18] K. Shoens, A. Luniewski, P. Schwarz, J. Stamos and J. Thomas, "The Rufus System: Information Organization for Semi-structured Data", *Proc. 19th Int'l Conf. Very Large Data Bases*, Dublin, Ireland, Aug. 1993, 97-106.

[19] R. Soley and W. Kent, "The OMG Object Model", in *Database Challenges in the 1990s*, Won Kim (ed.), Addison-Wesley/ACM, (in preparation).

[20] B. Stroustrup, *The C++ Programming Language (2nd Ed.)*, Addison-Wesley, 1991.

Appendix

Here are simple versions of the OdeFS interface functions for the `person` class defined in Figure 4. For simplicity, they ignore various overflow situations. The function `getStrFld`, scans a character array for a line starting with a specified field name, and if found, copies the rest of the line into another character array, and returns 1. The function `ofsIsClass` returns 1 if an object is of the indicated class, or of a class derived from it.

```
typedef persistent person* personP;
typedef persistent void*   voidP;
class OfsExtFcn_person: public OfsExtFcn {
 public:
  void  ofsName(const personP, char*, int);
  int   ofsRead(const personP, char*, int);
  int   ofsWrite(personP, const char*, int);
  voidP ofsLookupRef(const personP, char*);
  int   ofsReplaceRef(personP, char*, voidP);
  personP ofsCreate(const char*, int);
  personP ofsFind(const char*);
  };

// Copy person's file name to buff.
void OfsExtFcn_person::ofsName(
          const personP p, char* buff, int)
{ const char* s = p->getName();
  for (char* t = buff; *s != '\0'; s++)
    if (!isspace(*s)) *t++ = *s;
  *t = '\0';
}

// Put person's ofile in buff, return length.
int OfsExtFcn_person::ofsRead(
      const personP p, char* buff, int mlen)
{ ostrstream sb(buff, mlen);
  sb << "Name\t" << p->getName() << endl;
  sb << "Address\t" << p->getAddr() << endl;
  if (p->getSpouse() != 0)
    sb << "Spouse\t" <<
        p->getSpouse()->getFname() << endl;
  return sb.pcount();
}

// Update person object from data in buff.
int OfsExtFcn_person::ofsWrite(personP p,
          const char* buff, int)
{ char  name[256], addr[256];
  if (getStrFld("Name", buff, name)) {
    if (name[0] == '\0') {
      ofsErrMsg("Name required");
      return 0;
    }
    p->setName(name);
  }
  if (getStrFld("Address", buff, addr))
    p->setAddr(addr);
  return 1;
}

// Create and return new person object.
personP OfsExtFcn_person::ofsCreate(
          const char* buff, int len)
{ char  name[256];
  if (!getStrFld("Name", buff, name)) {
    ofsErrMsg("Name required");
    return 0;
  }
  personP p = pnew person(name);
  if (!ofsWrite(p, buff, len))
    { pdelete p; return 0; }
  return p;
}

// Find person matching simple query.
personP OfsExtFcn_person::ofsFind(
          const char* query)
{ if (memcmp(query, "Name=", 5) == 0) {
    const char *name = query+5; personP p;
    for (p in person)
      suchthat(strcmp(p->getName(),name) == 0)
        return p;
  }
  if (memcmp(query, "Address=", 8) == 0) {
    const char *addr = query+8; personP p;
    for (p in person)
      suchthat(strcmp(p->getAddr(),addr) == 0)
        return p;
  }
  return 0;
}

// Return referenced object.
voidP OfsExtFcn_person::ofsLookupRef(
          const personP p, char* name)
{ if (strcmp(name, "spouse") == 0)
    return p->getSpouse();
  return 0;
}

// Change ref'd object, return 1 if okay.
int OfsExtFcn_person::ofsReplaceRef(
          personP p, char* name, voidP xp)
{ if (strcmp(name, "spouse") == 0
   && ofsIsClass(p, "person")) {
    p->setSpouse((personP)xp);
    return 1;
  }
  return 0;
}
```